IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received August 16, 2021, accepted August 27, 2021, date of publication August 31, 2021, date of current version September 10, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3109491

FPGA Acceleration of 3D FDTD for Multi-Antennas Microwave Imaging Using HLS

MOHAMMAD AMIR MANSOORI[1], (Member, IEEE), PAN LU[2], AND MARIO R. CASU[1], (Senior Member, IEEE)
[1]Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Torino, Italy
[2]Department of Engineering, King's College London, London WC2R 2LS, U.K.

Corresponding author: Mohammad Amir Mansoori (mohammadamir.mansoori@polito.it)

This work was supported by the ElectroMagnetic imaging for a novel genERation of medicAL Devices (EMERALD) Project (website: http://www.msca-emerald.eu/) by the European Union's Horizon 2020 Research and Innovation Program through the Marie Skłodowska-Curie Grant under Agreement 764479.

ABSTRACT Microwave Imaging (MI) for biomedical applications has attracted attention due to its harmless radiation compared to X-ray or MRI. One of the commonly used computing methods in MI is Finite Difference Time Domain (FDTD), which is executed several times in iterative loops, hence resulting in a high execution time. Although several hardware accelerators for FDTD have been recently introduced, they are not specifically designed for MI applications. In particular, only simple absorbing boundary conditions have been investigated, and the impact of dispersive materials on FDTD has not been considered. In this paper, we propose a multi-FPGA accelerator for 3D FDTD that is integrated in an MI algorithm, with Convolutional Perfectly Matched Layer (CPML) boundary conditions and an exact model for dispersive materials. By using High Level Synthesis (HLS), we obtain an optimized hardware accelerator that uses an efficient blocking method to reduce the data transfer time between external and local memories. We propose two alternative architectures that trade off performance and resource usage. In addition, our code, being developed at a high level, can also be run on GPUs whenever necessary. The results show that our multi-FPGA accelerator is superior to three similar GPU-based designs in terms of execution time and power consumption.

INDEX TERMS FPGA, HLS, FDTD, hardware acceleration, microwave imaging.

I. INTRODUCTION

Microwave Imaging (MI) uses microwaves emitted and captured by several antennas to create an image of the inner dielectric profile of an object. It has attracted attention among biomedical researchers due to its low-cost, non-ionizing and non-invasive characteristics. Medical diagnosis in MI is based on the contrast between the dielectric properties of normal and anomalous tissues [1].

MI solves the electromagnetic *inverse-scattering* problem, which is inherently non-linear and makes the reconstruction a challenging task. Although approximate linear methods have been used [2]–[4], they have limited accuracy especially when the object is highly heterogeneous. More accurate, non-linear approaches solve the inverse problem by updating the dielectric estimation iteratively, which results in a high execution time. Fig. 1 shows a general diagram of the iterative non-linear inverse scattering reconstruction for medical imaging. Starting from an initial guess of the dielectric profile, the *forward solver*, often implemented using the Finite Difference Time Domain (FDTD) approach [5], computes the electromagnetic fields. The output of the forward solver is compared with the actual microwave measurements and, based on the error, the dielectric profile is updated with a specific inverse scattering algorithm. In this work, we used the DBIM-TwIST algorithm for this inversion [6] and FDTD as forward solver.

The high execution time comes from FDTD, for which several hardware accelerators have been proposed, as covered in Sec. II. Although a GPU implementation is a natural choice, depending on the complexity of the problem—primarily number of elements in the volume and number of antennas—a 3D image reconstruction can still take hours to finish. This motivated us to design an alternative hardware accelerator for the MI algorithm developed by Miao and Kosmas [6].

The MI algorithm was originally coded in MATLAB with the forward part accelerated by a Tesla GPU using an

The associate editor coordinating the review of this manuscript and approving it for publication was Liang-Bi Chen.

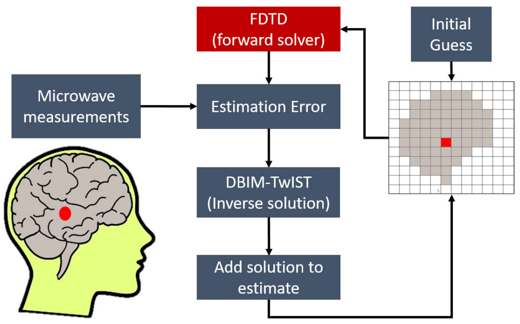


FIGURE 1. General diagram of a non-linear image reconstruction iterative algorithm in MI, with the compute-intensive FDTD step.

efficient commercial FDTD software library, *Acceleware* [7]. This code keeps the GPU fully busy by efficiently parallelizing the execution on the number of elements in the 3D volume. Since there is no room left to further parallelize on the number of antennas, the GPU code of the FDTD is executed sequentially for each of them. In MI systems with tens of antennas [4], leveraging instead the antenna parallelism could be the key to reducing the overall execution time from hours to minutes.

Although one obvious solution is to parallelize the execution on many GPUs, this is impractical for various reasons, including cost, form factor, and overall power consumption. Instead, an implementation on a *Multi-FPGA platform* can offer an equivalent performance at a fraction of cost and power, not to mention the much more manageable size and weight. A GPU implementation, however, can still outperform FPGA platforms with limited capacity.

For these reasons, we developed a 3D FDTD accelerator using a high-level approach, so that the code can be both implemented in FPGA using a High-Level Synthesis (HLS) flow, or easily changed to be executed in a GPU. This is possible because current FPGA design flows accept portable C/C++ high-level descriptions, which are enriched with specific directives, termed *pragmas*, for generating the desired Register-Transfer Level (RTL) code for FPGA hardware implementation. This high-level approach allows to explore the design space by changing the pragmas in a more efficient way compared to RTL design. In [8], a comprehensive analysis and the implications of using several HLS optimization transformations (including the HLS pragmas) have been presented for High-Performance Computing applications. In [9] and [10], the efficiency and performance of HLS-based design space exploration are explored. In [11], a fast HLS simulator is introduced to accelerate the hardware simulation process, and in [12] and [13], new methodologies are proposed for the optimum selection of HLS directives.

One of the design challenges for FDTD is the optimization of the memory access to a large amount of data, which is complicated by the relatively low amount of on-chip memory. While many previous publications considered a less challenging 2D FDTD, we focus instead on a full 3D implementation. To cope with the memory access issues in 3D FDTD, the few previous works on the subject use specific blocking methods to read blocks of data from the external memory, which is an approach that we also use in this work.

However, previous 3D FDTD works use simplified Periodic Boundary Conditions (PBC), which is not an accurate approach for some MI problems, but simplifies the hardware design. Improving the accuracy calls for more appropriate, but more difficult to implement in hardware, boundary conditions like Convolutional Perfectly Matched Layer (CPML), which is used in few works and in a limited way. Finally, previous works on 3D FDTD do not consider *dispersive* materials, which leads to more complex equations with dependencies that result in less straightforward parallelization. To the best of our knowledge, we are the first to propose a full-fledged 3D FDTD in FPGA that implements both CPML boundary conditions in all directions and uses an exact model for dispersive materials. We propose two possible FPGA implementations that use a different amount of on-chip memory, which creates a trade off between performance and resource usage.

In summary, the following is the list of our contributions:

- We propose the first FPGA accelerator for 3D FDTD integrated in an MI algorithm for medical applications.
- This is the first 3D FDTD accelerator to fully model dispersive materials, which makes the FPGA design more challenging.
- The CPML boundary conditions for 3D FDTD are used for all directions in contrast to previous accelerators designed with a high level approach that either do not consider CPML or consider periodic structures with CPML conditions only for one direction.
- Two hardware architectures with different characteristics are proposed and their pros and cons are analyzed.
- The entire hardware is designed using a High Level Synthesis (HLS) tool and several specific hardware optimization methods are used to design an efficient hardware.
- Both single- and multi-FPGA platforms are analyzed that can be used to accelerate FDTD with multiple antennas.
- The GPU implementation derived from our 3D FDTD code has a comparable performance with a commercial GPU implementation.

In the remainder of the paper, we discuss the related work in Sec. II and the principles of FDTD for MI in Sec. III, present the FPGA hardware accelerator in Sec. IV and related results in Sec. VI. In Sec. VII we discuss the challenges of the design and an overview of the solutions, and finally, we conclude in Sec. VIII.

II. RELATED WORK

Several FDTD accelerators proposed in the literature are based on GPUs. For instance, in [14] an implementation based on CUDA associates each thread to a cell in the FDTD grid and obtains the same accuracy of the CPU design with a speed-up ratio proportional to the grid size. Other GPU-accelerated versions of FDTD are proposed in [15]–[17].

High power consumption of GPUs draws attention to FPGA implementations. In [18] an FPGA accelerator for 2D FDTD is designed using OpenCL for two hardware platforms. The authors apply several OpenCL pragmas to create deeply pipelined loops. However, they do not consider the impact of boundary conditions. The FDTD hardware accelerator in [19] uses a HLS tool called MaxCompiler developed by Maxeler technologies. In their work, the authors investigate different boundary conditions, including PBC and CPML. However, their design can be used only for periodic structures where one can model the entire simulation space by a single periodic cell. In this cell, CPML conditions are applied only to the top and bottom boundaries, and PBC conditions are used for the other four boundaries. In contrast, we consider CPML in all directions as required by the MI application, at the cost of a much more complex design.

Takei *et al.* present an OpenCL-based design for 2D FDTD on FPGA [20]. To reduce the global memory access, they used an overlapped tiling method that can locally store small blocks of data. Despite lower power consumption compared to GPU, the processing time could not be reduced for large grid sizes. Waidyasooriya *et al.* in [21] extend the work in [20] to 3D FDTD by pipelining multiple FDTD iterations. Although they achieve better performance than CPU- or GPU-based designs, they only consider periodic structures for the boundary conditions. In addition, they simplify the FDTD update equations by ignoring the polarization current, hence reducing the required memory bandwidth. Recently, in [22] an FPGA design for 3D FDTD that considers CPML boundary conditions has been proposed, although the authors do not consider the impact of dispersive materials and polarization currents. In addition, they use Verilog to design their hardware at RTL, which increases the design and development time compared to the HLS-based design and makes less efficient the design space exploration.

FDTD can be seen as a *Stencil* computation. In stencils, the elements of a multi-dimensional grid are updated iteratively based on the neighbouring cells using a fixed pattern. The main bottleneck in both GPU and FPGA designs for stencil computation is the data transfer time between global and local memories. The common approach to alleviate this problem is to use *spatial* or *temporal* blocking. In the former, a spatial block of data is stored in on-chip memory to reduce the access time, and in the latter, different time steps are pipelined for further parallelization. Regarding stencil acceleration in FPGAs, there have been extensive research works in recent years. In [23], Waidyasooriya *et al.* extended their previous FPGA accelerator to a general stencil computation by increasing the degree of parallelism. In addition to pipelining multiple iterations, they could compute multiple grid cells in parallel. However, they did not report results for 3D FDTD with complex boundary conditions like CPML. In [24], another FPGA design for 3D stencils using OpenCL uses a combination of spatial and temporal blocking methods. In [25], [26], memory and power performance of FPGA accelerators for general stencils have been investigated. Other successful designs for stencil acceleration have been presented in [27] and [28]. Although some of the above works have considered FDTD as a benchmark for stencil computation, they analyzed simplistic scenarios that cannot be adapted to the special requirements of FDTD as used in MI. For example, simple boundary conditions like Dirichlet [25], [26], [28] or PBC [24] cannot be used in MI. *Polybench*, a benchmark suite used in some stencil accelerators, like [27], does not include a full 3D FDTD as only the Transverse Electric (TE) mode is considered (other directions of the fields are ignored). In addition, not modeling dispersive materials as done in [23] can improve the hardware acceleration (e.g., with deeper pipelining on time iterations), but cannot be done in MI applications.

III. FDTD IN MICROWAVE IMAGING

A. BACKGROUND

The main equations in FDTD for MI are the time-domain Maxwell equations for dispersive and lossy materials:

$$\nabla \times H = \frac{\partial D}{\partial t} + \sigma_e E + J_S \tag{1}$$
$$D(\omega) = \epsilon(\omega)E(\omega) \tag{2}$$
$$-\nabla \times E = \frac{\partial B}{\partial t} + \sigma_m H + M_S \tag{3}$$
$$B(\omega) = \mu(\omega)H(\omega) \tag{4}$$

H and E are magnetic and electric fields; B and D are magnetic and electric flux densities; M_S and J_S are magnetic and electric current densities (zero in the following); σ_m and σ_e are magnetic and electric conductivity; μ is magnetic permeability and ϵ is electric permittivity.

Dielectric materials in MI have $\mu = \mu_0$, the permeability of free space. Thus, $B = \mu_0 H$ and (3)-(4) can be merged into one equation ($-\nabla\times E = \mu_0 \frac{\partial H}{\partial t} + \sigma_m H + M_S$). On the contrary, the frequency dependency of ϵ in dispersive materials must be modeled, typically with the Debye model that we also use here. Taking this into account, Eqn. (2) can be written as: $D = \epsilon_0 E + P$, in which ϵ_0 is the free-space permittivity and P is the *polarization* vector that is proportional to E. Polarization current, J_P, is the time derivative of P: $J_P = \frac{\partial P}{\partial t}$.

FDTD solves the equations above based on finite difference approximations. The 3D volume is divided into cuboids called Yee cells [29] in such a way that each magnetic field is surrounded by four electric fields and vice versa. This results in two main *update* equations for electric and magnetic fields, respectively. Compared to the simpler case of non-dispersive materials, however, an additional variable accounting for the polarization current (J_P) appears in the update equation for the electric field only. The FDTD algorithm for a Debye model repeats the following two steps at each time step [5]:

1) update magnetic fields;
2) for each electric field component:

a) update electric fields and store them in a new variable E';
b) update polarization currents based on the new and old values of electric fields;
c) update the old fields with the new ones ($E = E'$).

B. BOUNDARY CONDITIONS: CPML

FDTD is used to obtain the propagation of the electromagnetic waves in the simulation space. Due to the finite size of this space, the propagation must be terminated in the ‘‘Boundary Regions’’. Therefore, the update equations in these regions must be modified by adding proper boundary conditions. Dirichlet conditions consider that the fields in the boundaries are zero, while PBC considers the fields to be repeated after a fixed number of cells. These conditions will create unwanted reflections from the boundary regions towards the inner simulation space. CPML is a more complex boundary condition that eliminates these reflections by letting the propagation be absorbed in the boundary region.

To better understand the following description of the FDTD code and the role played by the boundary conditions the exemplifications in Fig. 2 are helpful. The figure shows all the cells and identifies two boundary planes for each *direction* (x, y, z) using a different coloring. The figure refers to the magnetic field and shows that, for instance, the components H_x and H_z must be updated in the two planes of the y direction (called *front* and *back faces* in the following), while the components H_y and H_z must be updated in the two planes of the x direction (*left* and *right faces*), and so on. A similar figure can be used for the electric field.

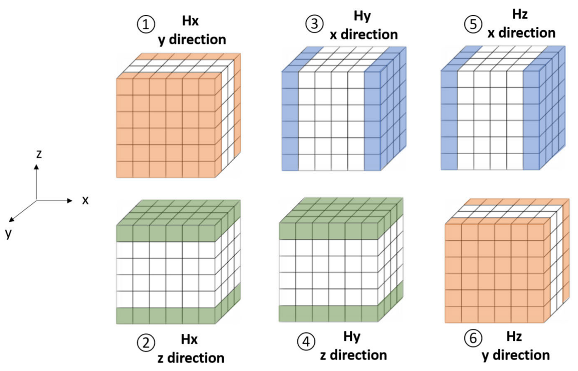


FIGURE 2. Boundary regions for *H* field in 3D FDTD.

Algorithm 1: 3D FDTD Pseudo-Code

```
for $s \in$ Antennas do // Loop over antennas
    for $c \in$ Domain Cells $\cup$ Boundary Cells do // Initialize at t=0
        $E_{\{x,y,z\}}(s, c, t = 0) = 0,\ H_{\{x,y,z\}}(s, c, t = 0) = 0$
    // From now on (s,c,t) omitted for readability
    for $t = 1$ to $T_{\max}$ do // Loop over time steps
        // Update H: loop over all cells
        for $c \in$ Domain Cells $\cup$ Boundary Cells do
            $H_x = a_{hx}H_x + b_{hy}(E_y^{+x} - E_y^c) + b_{hz}(E_z^{+x} - E_z^c)$
            $H_y = a_{hy}H_y + b_{hx}(E_x^{+y} - E_x^c) + b_{hz}(E_z^{+y} - E_z^c)$
            $H_z = a_{hz}H_z + b_{hx}(E_x^{+z} - E_x^c) + b_{hy}(E_y^{+z} - E_y^c)$
        // Update H boundary: loop over boundary cells
        for $c \in$ Boundary Cells of y direction, front face do
            $\Psi_{Hxy} = c_{hy1}\Psi_{Hxy} + c_{hy2}(E_z^{+y} - E_z^c),\ H_x \mathrel{+}= d_{hx}\Psi_{Hxy}$
            $\Psi_{Hzy} = c_{hy1}\Psi_{Hzy} + c_{hy2}(E_x^{+y} - E_x^c),\ H_z \mathrel{+}= d_{hz}\Psi_{Hzy}$
        for $c \in$ Boundary Cells of y direction, back face do
            // see footnote[a]
        for $c \in$ Boundary Cells of x direction, left face do
            $\Psi_{Hyx} = c_{hx1}\Psi_{Hyx} + c_{hx2}(E_z^{+x} - E_z^c),\ H_y \mathrel{+}= d_{hy}\Psi_{Hyx}$
            $\Psi_{Hzx} = c_{hx1}\Psi_{Hzx} + c_{hx2}(E_y^{+x} - E_y^c),\ H_z \mathrel{+}= d_{hz}\Psi_{Hzx}$
        for $c \in$ Boundary Cells of x direction, right face do
            // see footnote[a]
        for $c \in$ Boundary Cells of z direction, top face do
            $\Psi_{Hxz} = c_{hz1}\Psi_{Hxz} + c_{hz2}(E_y^{+z} - E_y^c),\ H_x \mathrel{+}= d_{hx}\Psi_{Hxz}$
            $\Psi_{Hyz} = c_{hz1}\Psi_{Hyz} + c_{hz2}(E_x^{+z} - E_x^c),\ H_y \mathrel{+}= d_{hy}\Psi_{Hyz}$
        for $c \in$ Boundary Cells of z direction, bottom face do
            // see footnote[a]
        // Update E: loop over all cells
        for $c \in$ Domain Cells $\cup$ Boundary Cells do
            $E'_x = a_{ex}E_x + b_{ey}(H_y^c - H_y^{-x}) + b_{ez}(H_z^c - H_z^{-x})$
            $E'_y = a_{ey}E_y + b_{ex}(H_x^c - H_x^{-y}) + b_{ez}(H_z^c - H_z^{-y})$
            $E'_z = a_{ez}E_z + b_{ex}(H_x^c - H_x^{-z}) + b_{ey}(H_y^c - H_y^{-z})$
            $E'_x \mathrel{+}= c_P J_{Px},\ E'_y \mathrel{+}= c_P J_{Py},\ E'_z \mathrel{+}= c_P J_{Pz}$
        // Update E boundary: loop over boundary cells
        for $c \in$ Boundary Cells of y direction, front face do
            $\Psi_{Exy} = c_{ey1}\Psi_{Exy} + c_{ey2}(H_z^c - H_z^{-y}),\ E'_x \mathrel{+}= d_{ex}\Psi_{Exy}$
            $\Psi_{Ezy} = c_{ey1}\Psi_{Ezy} + c_{ey2}(H_x^c - H_x^{-y}),\ E'_z \mathrel{+}= d_{ez}\Psi_{Ezy}$
        for $c \in$ Boundary Cells of y direction, back face do
            // see footnote[b]
        for $c \in$ Boundary Cells of x direction, left face do
            $\Psi_{Eyx} = c_{ex1}\Psi_{Eyx} + c_{ex2}(H_z^c - H_z^{-x}),\ E'_y \mathrel{+}= d_{ey}\Psi_{Eyx}$
            $\Psi_{Ezx} = c_{ex1}\Psi_{Ezx} + c_{ex2}(H_y^c - H_y^{-x}),\ E'_z \mathrel{+}= d_{ez}\Psi_{Ezx}$
        for $c \in$ Boundary Cells of x direction, right face do
            // see footnote[b]
        for $c \in$ Boundary Cells of z direction, top face do
            $\Psi_{Exz} = c_{ez1}\Psi_{Exz} + c_{ez2}(H_y^c - H_y^{-z}),\ E'_x \mathrel{+}= d_{ex}\Psi_{Exz}$
            $\Psi_{Eyz} = c_{ez1}\Psi_{Eyz} + c_{ez2}(H_x^c - H_x^{-z}),\ E'_y \mathrel{+}= d_{ey}\Psi_{Eyz}$
        for $c \in$ Boundary Cells of z direction, bottom face do
            // see footnote[b]
        // Update $J_P$: loop over all cells
        for $c \in$ Domain Cells $\cup$ Boundary Cells do
            $J_{P\{x,y,z\}} = s_P J_{P\{x,y,z\}} + Q_P(E'_{\{x,y,z\}} - E_{\{x,y,z\}})$
            $E_{\{x,y,z\}} = E'_{\{x,y,z\}}$
        for $c \in$ Antenna Cells do // Add antenna source
          signals:
            $E_z(s, c, t) = \mathrm{Source}(s, c, t)$
```

[a]Same equations of previous loop with new $\hat{\Psi}_H$ and $\hat{c}_h$ variables
[b]Same equations of previous loop with new $\hat{\Psi}_E$ and $\hat{c}_e$ variables

C. FDTD PSEUDO-CODE

Each FDTD update equation can be written in a general form like the following equation for H_x:

$$H_x = a_{hx}H_x + b_{hy}(E_y^{+x} - E_y^c) + b_{hz}(E_z^{+x} - E_z^c) + d_{hx}(\Psi_{Hxy} + \Psi_{Hxz}). \quad (5)$$

E^c and E^{+x} are the electric field of the central cell being calculated and of the next cell, respectively; Ψ_{Hxy}, Ψ_{Hxz} are used in the boundaries only and can be considered to be zero in the main cells. All the other terms are constant with a spatial dependency. When computing the cells in the boundary layers in y direction, Ψ_{Hxz} is zero and when updating the cells in the boundaries of z direction, Ψ_{Hxy} is zero. In the overlay of boundary cells in y and z directions, both terms are present. Therefore, the update equation for H_x can be divided into different regions including the main cells, boundary cells of y direction (front and back) and boundary cells of z direction (top and bottom). Separate loops must be considered for each region to obtain the final output.

These separate loops are described by the FDTD pseudo-code in Alg. 1. Notice the difference between *Update H* and *Update E* equations due to the polarization currents $J_{P\{x,y,z\}}$. E^c, H^c are the electric and magnetic fields of the central cell being calculated; (E^{+x}, E^{+y}, E^{+z}) and (H^{-x}, H^{-y}, H^{-z}) are the electric and magnetic fields of the next and the previous cell, respectively, in (x, y, z) directions. Except variables $H, E, J_P, \Psi_H, \Psi_E$, all the other terms are constant with a spatial dependency, which requires a significant amount of memory. For more details on FDTD please refer to [5]. Note that the dependency on time step (t) and antenna index (s) in Alg. 1 is omitted whenever needed to improve readability. In addition, even though the antenna source signals can be added in any direction, in our design the antennas emit an electric field in the z direction.

As Alg. 1 clearly shows, the outer loop can be easily *unrolled* and assigned to different parallel Compute Units (CUs), each in charge of one antenna. In the next two sections, we focus first on the design and optimization of a single CU and its implementation on one FPGA, then we focus on the multi-FPGA implementation of a multi-CU system.

IV. FPGA DESIGN OF AN FDTD COMPUTE UNIT

We validated our initial C++ design in terms of accuracy against the Acceleware commercial code. Both codes use 32-bit Floating-Point (FP) data for all the variables in Alg. 1. Note that computing precision is critical in MI iterative algorithms, as the errors tend to accumulate and lead to inexact solutions. This is why fixed-point data type, which would certainly lead to higher computation speed, cannot be used in this case.

Although we easily converted our initial code to RTL for FPGA implementation using HLS tools,[1] the estimated performance (latency in number of clock cycles times the estimated clock period) was worse than the GPU one in Acceleware. To enhance the performance, we adopted various hardware optimization strategies, which consisted in the use of specific HLS *pragmas* and some modifications to the original C++ code. Although the code can be easily adapted to different FPGAs from different vendors, we focused our optimizations and experiments on a Xilinx target. Therefore, from now on, we often refer to *Vivado HLS* and *Vivado* as the tools for HLS development and implementation, respectively. It is important to note that the design goal in our hardware accelerator is to minimize the total latency of the FDTD computation, because this reduces the overall execution time of the MI iterative algorithm. This can be achieved by applying the HLS optimization strategies described in the following.

Fig. 3 presents a schematic representation of the HLS code for a single CU, in which each block is a function that corresponds to an update equation in Alg. 1. The J_P equations, which are separate in Alg. 1, are merged with the Update E and E boundary blocks in Fig. 3 to avoid rereading the E fields from the external memory. Table 1 summarizes the HLS optimization techniques used in the hardware design and the functions in which they are used, as explained thoroughly in the next subsections. The *top-level function* denotes the function that contains the loop over the time steps in Alg. 1.

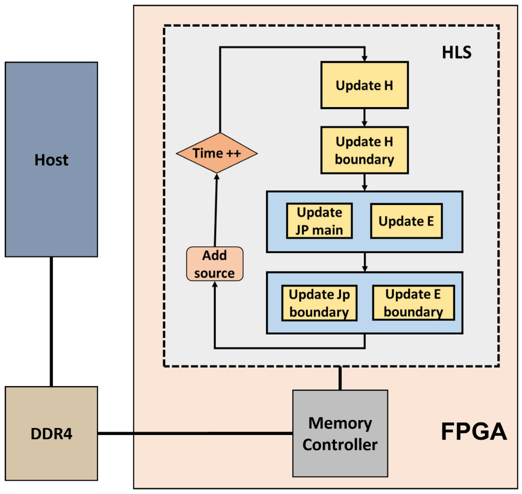


FIGURE 3. FDTD CU design in HLS for a single FPGA.

TABLE 1. HLS hardware optimization strategies for a FDTD CU.

Method	functions
Blocking	Update E and Update H
Merge JP	Update E and E boundary
Loop merge	Update E boundary and H boundary
Storage for Boundaries	Update H boundary and/or Update E boundary
Storage for constant coefficients	top-level function
Loop pipeline	all
function inline	all

Vivado HLS synthesizes the C++ code and generates the RTL code of an IP block that is integrated in the Vivado implementation flow with the additional blocks in Fig. 4. These include a memory controller, two reset blocks, and an AXI interconnect block that connects to the memory controller the IP interfaces, all compliant with the AXI standard.

A. TWO ARCHITECTURES: LARGE AND SMALL

For a more flexible design, we propose two hardware architectures, termed *Large* and *Small,* which use a different number of AXI I/O interfaces and a different amount of local on-chip memory for storing magnetic or electric fields in the boundary regions. In the large design more computing resources and more AXI interfaces are used, which results both in a lower number of CUs that can be implemented in a single FPGA but also in a lower latency per CU compared

[1]Both the development tools and the advanced target FPGAs nowadays support the synthesis of FP arithmetic to hardware.

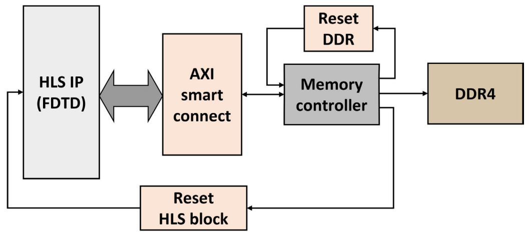


FIGURE 4. Detailed view of the CU design in Vivado.

to the Small design in which less resources and interfaces are used. Hence, depending on the number of CUs over which a designer wishes to parallelize the computation and the number of available FPGAs, either the Large or the Small design is the preferred choice in terms of execution time, as shown later.

To avoid performance bottlenecks in the Large design, 18 AXI interfaces are needed, as shown in Fig. 5, to guarantee a concurrent access to all the data arrays needed during the computation. Note that the aggregate bandwidth is still compatible with the external DRAM specifications, as we show later. Since in our target FPGA each memory controller can handle only up to 16 ports, one CU needs two memory controllers, which is possible in large FPGAs with multiple external DRAM memory banks.

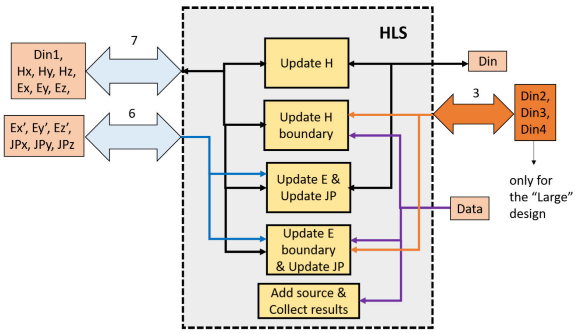

Din1, Hx, Hy, Hz, Ex, Ey, Ez,
7
Ex', Ey', Ez', JPx, JPy, JPz
6
HLS
Update H
Update H boundary
Update E & Update JP
Update E boundary & Update JP
Add source & Collect results
Din
3
Din2, Din3, Din4
only for the "Large" design
Data

FIGURE 5. Details of the interfaces of the CU for Small and Large designs.

In the Small design, on the other hand, we reduced the number of AXI ports to 15 by removing three AXI ports (Din2, Din3, Din4) (orange box in Fig.5) and using a shared port (Din1) instead. This reduction of AXI interfaces makes it possible to reduce the resource usage as shown in Sec. IV-C and Sec. VI-B. As a result, the data in the Small design is routed to some of the blocks in Fig. 5 via shared interfaces, at the cost of a lower performance. However, by using lower resources, we can fit more CUs into a single large FPGA with multiple memory controllers, each handling one single CU, hence increasing the overall throughput per single FPGA.

Table 2 shows how the variables in Alg. 1 are mapped to the AXI ports according to the design version. Note that Din2-4 only exist in the Large design. In Tab 2, b_h_xy is the combination of b_{hx} and b_{hy} in one single array, and similarly b_e_xy for b_{ex} and b_{ey}. Ψ_{E*} and Ψ_{H*} associated to port Din1 refer to all the variables of that type in Alg. 1.

TABLE 2. Description of AXI interfaces.

AXI ports	Design	Variables (refer to Alg. 1)
Data	Small/Large	$c_{h1}, c_{h2}, c_{e1}, c_{e2}, Qp, source$
Din	Small/Large	b_h_xy, b_e_xy
Din1	Small	$\Psi_{E*}, \Psi_{H*}, b_{hz}, b_{ez}$
	Large	$\Psi_{E*}, b_{hz}, b_{ez}, \Psi_{Hxy}, \Psi_{Hyx}, \Psi_{Hxz}$
Din2	Large	$\Psi_{Hzy}, \Psi_{Hzx}, \Psi_{Hyz}$
Din3	Large	$\hat{\Psi}_{Hxy}, \hat{\Psi}_{Hyx}, \hat{\Psi}_{Hxz}$
Din4	Large	$\hat{\Psi}_{Hzy}, \hat{\Psi}_{Hzx}, \hat{\Psi}_{Hyz}$
Ex, Ey, Ez	Small/Large	E_x, E_y, E_z
Ex′, Ey′, Ez′	Small/Large	E'_x, E'_y, E'_z
Hx, Hy, Hz	Small/Large	H_x, H_y, H_z
JPx, JPy, JPz	Small/Large	J_{Px}, J_{Py}, J_{Pz}

In the following, we explain the optimization methods listed in Table 1.

B. BLOCKING METHOD AND MERGING OF J_P UPDATE EQUATIONS

To optimize the memory access, we use a method similar to the spatial blocking used in stencils. Fig. 6(a) shows the general approach using shift registers as local memory in a 2D stencil with dimensions X and Y. The stencil moves from left to right until it reaches the end of a row and moves one row downward. To compute the current cell C, the four surrounding cells (N,E,S,W) are needed. As the stencil moves, a new cell is written in the head of a shift register and used immediately as "new" S cell, while the "old" N cell is removed from the tail of the shift register. The shift register holds the last $2X + 1$ cells from the grid (darker orange cells).

There are significant differences between a generic 2D stencil and a 3D FDTD. As shown in Fig. 6(b), updating the magnetic field in a cell requires the electric field in the current and *next* positions, while updating the electric field requires the magnetic field in the current and *previous* positions. Therefore, instead of the 5-cell stencil pattern, we need 3 cells for H update and 3 cells for E update. For a 2D FDTD, this would require two shift registers (for H and E) each of size $X + 1$, but for a 3D FDTD, the size becomes $XZ + 1$.

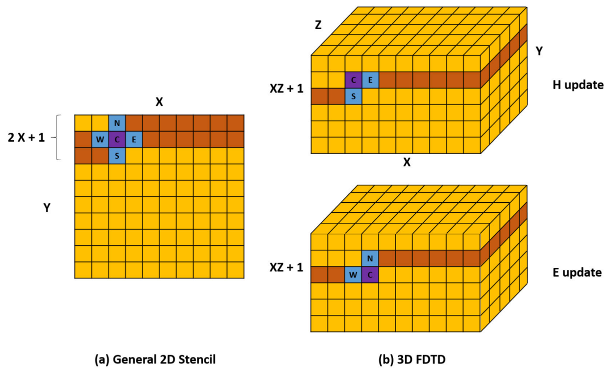


FIGURE 6. Blocking method for FDTD and its difference with a general stencil.

In addition, since in 3D FDTD the three components of E and H fields need each a separate shift register, the number of resources in FPGA is severely impacted.

Algorithm 2: Pseudo-Code for Update E With Blocking and JP Merge

```
Update E:
$s = 0$; $size = (X - 1)(Y - 1) + 1$;
for $k$=Z-1;k>1;k-- do // Process XY planes in Z direction
    for Domain Cells ∪ Boundary Cells in plane $k$ do
        if Boundary Cells and Use local storage for H then
            $BF = H$;// Local storage in BRAM
        Blocking:
        // Store new XY plane in memory
        $Hram_{\{x,y,z\}}[s{+}{+}] = H_{\{x,y,z\}}[k]$;
        #pragma resource Hram RAM_2P_LUTRAM
        if $s = size$ then // Hram=full
            $s = 0$;
        if ($k < Z - 1$) then // ignore first plane (Z-1)
            // Process old XY plane (while reading
            new)
            $H^{c}_{\{x,y,z\}} = Hram_{\{x,y,z\}}[s]$;
            $H^{-y}_{\{x,z\}} = Hram_{\{x,z\}}[(s + X - 1)\%size]$;
            $H^{-z}_{\{x,y\}} = Hram_{\{x,y\}}[(s + size - 1)\%size]$;
            $H^{-x}_{\{y,z\}} = Hram_{\{y,z\}}[(s + 1)\%size]$;
            Main Update E (for plane $k + 1$):
            $E'_x = UpdateE(E_x, H^c_y, H^{-x}_y, H^c_z, H^{-x}_z, J_{Px})$;
            $E'_y = UpdateE(E_y, H^c_x, H^{-y}_x, H^c_z, H^{-y}_z, J_{Py})$;
            $E'_z = UpdateE(E_z, H^c_x, H^{-z}_x, H^c_y, H^{-z}_y, J_{Pz})$;
            Merge JP update (for plane $k + 1$):
            if Domain Cells then
                $J_{P\{x,y,z\}} = s_P J_{P\{x,y,z\}} + Q_P(E'_{\{x,y,z\}} - E_{\{x,y,z\}})$;
                $E_{\{x,y,z\}} = E'_{\{x,y,z\}}$;
```

Therefore, instead of using shift registers with Flip-Flops or Look-Up Tables (LUTs), we use BRAMs to locally store the fields. Since BRAMs are dual-port SRAMs, however, it is not possible to read more than two values per clock cycle. Moreover, we cannot partition the arrays to overcome the two-port limitation, because the accessed elements are not always in the same partition. Therefore, we replicate the BRAMs multiple times to simultaneously access all the required cells. As we will see in Sec. VI, to balance the resource utilization it is possible, by means of the HLS resource allocation directive, to replace the BRAMs with LUTs arranged as distributed memories.

This local memory for blocking is represented by the *Hram* variable in Alg. 2. Note that, after a round of initialization with an initial plane in the z direction, the memory gets filled with a new plane while the computation happens on the previous plane. The concurrency of memory access to the new plane and computation on the old plane is a key factor to obtain a high computing throughput and so a low execution latency. Another local memory used only for the boundary field cells is *BF* in Alg. 2, which is described in detail in Sec. IV-C.

Another optimization consists in merging the loop that updates J_P with the other loops for E update and E-boundary update, previously shown as three separate loops in Alg. 1. For this purpose, we split the J_P update equations in two parts for the main domain and boundary cells. In Alg. 2 the part of the main cells is updated with the *if* condition in the last few lines.

Algorithm 3: Pseudo-Code for Update H-Boundary With Merging Boundary Loops and Local Storage

```
Update H:
{...
if Boundary Cells and Use local storage for E then
    $BF = E$;// Local storage in BRAM
...}
Update H-boundary:
if NOT use local storage for E then
    Change BF to E
#pragma loop merge
for Boundary cells in y direction, front face do
    $\Psi_{Hxy} = c_{hy1}\Psi_{Hxy} + c_{hy2}(BF^{+y}_z - BF^c_z),\ H_x \mathrel{+}= d_{hx}\Psi_{Hxy}$;
    $\Psi_{Hzy} = c_{hy1}\Psi_{Hzy} + c_{hy2}(BF^{+y}_x - BF^c_x),\ H_z \mathrel{+}= d_{hz}\Psi_{Hzy}$;
for Boundary cells in y direction, back face do
    $\hat{\Psi}_{Hxy} = \hat{c}_{hy1}\hat{\Psi}_{Hxy} + \hat{c}_{hy2}(BF^{+y}_z - BF^c_z),\ H_x \mathrel{+}= d_{hx}\hat{\Psi}_{Hxy}$;
    $\hat{\Psi}_{Hzy} = \hat{c}_{hy1}\hat{\Psi}_{Hzy} + \hat{c}_{hy2}(BF^{+y}_x - BF^c_x),\ H_z \mathrel{+}= d_{hz}\hat{\Psi}_{Hzy}$;
// 2 (x2) other loops for x and z directions
```

For space reasons, we do not show the pseudo-code for Update H, which uses the same blocking method of Alg. 2.

C. LOOP MERGE AND LOCAL STORAGE FOR BOUNDARIES

A key strategy to improve the FDTD performance is to move the accesses of the many array variables from the external DRAM to on-chip SRAMs. Due to the large number of cells, however, even for the largest FPGAs this strategy is applicable only to a subset of the variables. For this reason, we use on-chip memory only for the boundary elements, when possible. This is shown in Algs. 2-3 with local storage *BF* enabled or disabled with an *if* conditional statement.

The loops for the update equations for the six boundary regions in Fig. 2, which are separated in Alg. 1, can be merged in pairs according to the coloring scheme in Fig. 2. Alg. 3 shows how the loops in Update H-boundary are merged by using the *loop merge* HLS pragma. Note that the complete CPML boundary conditions require 6 loops in total for all the directions for each field (E or H), while in previous works, the simplified CPML boundary conditions require only 2 loops as they are used only for one direction.

Despite the loop merging in the H-boundary update, the problem of accessing the Ψ_{H*} arrays from the external memory several times remains intact. To maximize the execution speed of the merged loops, the arrays need to be accessed four times in parallel, so the four AXI ports Din1-4 almost entirely dedicated to them shown in Table 2 and in Fig. 5 (3 dedicated ports, Din2-4, and one shared port, Din1).

While the Large design leverages the simultaneous access through these ports, the Small one eliminates ports Din2-4. As discussed later in Sec. VI-B, in the Small design there is no benefit in using the local BRAMs to store the electric fields ($BF = E$), since the performance is limited by the serialized access to all the Ψ_{H*} arrays through one shared AXI port.

The loop merging for the Update E-boundary is shown in Alg. 4. Here we merged the part of the JP update equations related to the boundary cells, while the part related to the main cells is merged with the E update, as shown in Alg. 2.

Algorithm 4: Pseudo-Code for Update E-Boundary With Merging Boundary Loops and J_P Merge

```
Update E:
{...
if Boundary Cells and Use local storage for H then
    $BF = H$;// Local storage in BRAM
...}
Update E-boundary:
if NOT use local storage for H then
    Change $BF$ to $H$;
#pragma loop merge
for Boundary cells in y direction, front face do
    $\Psi_{Exy} = c_{ey1}\Psi_{Exy} + c_{ey2}(BF_z^c - BF_z^{-y}),\ E'_x \mathrel{+}= d_{ex}\Psi_{Exy}$;
    $\Psi_{Ezy} = c_{ey1}\Psi_{Ezy} + c_{ey2}(BF_x^c - BF_x^{-y}),\ E'_z \mathrel{+}= d_{ez}\Psi_{Ezy}$;
    Merge JP update, front face:
    $J_{P\{x,z\}} = s_P J_{P\{x,z\}} + Q_P(E'_{\{x,z\}} - E_{\{x,z\}})$;
    $E_{\{x,z\}} = E'_{\{x,z\}}$;
for Boundary cells in y direction, back face do
    $\hat{\Psi}_{Exy} = \hat{c}_{ey1}\hat{\Psi}_{Exy} + \hat{c}_{ey2}(BF_z^c - BF_z^{-y}),\ E'_x \mathrel{+}= d_{ex}\hat{\Psi}_{Exy}$;
    $\hat{\Psi}_{Ezy} = \hat{c}_{ey1}\hat{\Psi}_{Ezy} + \hat{c}_{ey2}(BF_x^c - BF_x^{-y}),\ E'_z \mathrel{+}= d_{ez}\hat{\Psi}_{Ezy}$;
    Merge JP update, back face:
    $J_{P\{x,z\}} = s_P J_{P\{x,z\}} + Q_P(E'_{\{x,z\}} - E_{\{x,z\}})$;
    $E_{\{x,z\}} = E'_{\{x,z\}}$;
// 2(×2) other loops for x and z directions
```

D. LOOP PIPELINE, FUNCTION INLINE, AND STORAGE FOR COEFFICIENTS

The last optimization strategies consist in setting directives to a) inline all the functions and b) pipeline the innermost loop in nested loops like those over all the cells in three dimensions. Both strategies significantly improve the Initiation Interval (II) of the loops, which is the distance in clock cycles between the starting of two consecutive iterations and corresponds to the inverse of the loop throughput, as discussed thoroughly in Sec. VI. A perfectly pipelined loop starts a new computation every clock cycle (II=1). In practice, dependencies between iterations prevent to obtain this goal for every loop. As for function inlining, it allows for further resource sharing when this does not impact performance and allows for optimization across function hierarchies. Alg. 5 shows how the pragmas associated with these strategies are used in the update functions.

Algorithm 5: HLS Pragmas of Loop Pipelining and Function Inlining in 3D FDTD Algorithm

```
Update functions:
#pragma inline// Function is inlined
// Loop over all cells in (x,y,z) dimensions
for $k$=1 to Z do
    for $j$=1 to Y do
        for $i$=1 to X do // Innermost loop on x is pipelined
            #pragma pipeline
            ...
```

Finally, we locally store constant coefficients $a_{e\{x,y,z\}}$ and $d_{e\{x,y,z\}}$ in Alg. 1 in URAMs, which are large memories available in high-capacity Xilinx FPGAs. Since the coefficient values vary over the entire domain, they need a large amount of storage, but one AXI port is enough to load them during the initialization. By using both BRAMs and URAMs, we achieve a high utilization of local FPGA storage as shown in Sec. VI.

V. MULTI-FPGA IMPLEMENTATION

The possibility of unrolling the outermost loop in Alg. 1 and let multiple CUs work in parallel on different antennas, is hindered by the limited resources available in one FPGA. For example, the FPGA used in our experiments supports up to 3 CUs in the Small design and 2 CUs in the Large one. In particular, this FPGA contains three so-called Super Logic Regions (SLRs) and our fastest design uses one CU per SLR. Although we could place more CUs, the advantage of parallelism is countered by a) the slower memory access caused by the AXI ports sharing, and b) the slower clock frequencies caused by the routing congestion. The only chance to improve performance is to use multiple FPGAs.

We emphasize that the computations in each FPGA and for each antenna are independent and there is no need for data sharing between them. This straightforward parallelism simplifies the deployment over commercially available and energy-optimized multi-FPGA platforms. This is in contrast with the use of multiple GPUs, e.g. in High-Performance Computing (HPC) clusters, which are more expensive and less energy efficient for similar performance. To show this contrast and to demonstrate the advantages of using FPGAs over GPUs, we proposed the deployment of our FDTD accelerator on a Multi-FPGA platform.

Multi-FPGA platforms have become easily accessible, like for example the Amazon AWS EC2 F1 instances. The AWS platform has eight Xilinx UltraScale+ FPGAs, each connected to a multi-bank local DDR DRAM. The FPGAs are connected via the PCIexpress (PCIe) bus to an x86 host CPU, as shown in Fig. 7. The figure also shows how multiple CUs working on $3F$ or $2F$ antennas, depending on the design version with either 15 or 18 AXI ports, can be allocated to F FPGAs.

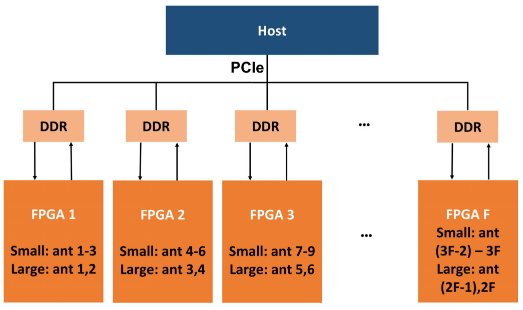


FIGURE 7. Multi-FPGA platform with F FPGAs for 3D FDTD acceleration.

The host code in the CPU coordinates the execution of the CUs in the FPGAs similarly to how the host code controls execution of multiple threads in a GPU. Initially the host transfers the inputs required by FDTD to all the local DDR memories. As we show in Sec. VI, the overhead for this initial PCIe transaction is negligible compared to the computing time.

The total execution time depends on the number of FPGAs, the number of antennas, and the design version. With A antennas and F FPGAs, depending on the maximum supported number of antennas n_A in each FPGA (2 in the Large version or 3 in the Small one) the relation between the total execution time (T_{tot}) and the time for each antenna (T_{ant}) is:

$$T_{tot} = \left\lceil \frac{A}{n_A \cdot F} \right\rceil \times T_{ant} \quad (6)$$

By using the AWS platform with 8 FPGAs, T_{tot} will be equal to T_{ant} for up to $8\times3 = 24$ or $8\times2 = 16$ antennas in the Small and Large designs, respectively. For a larger number of antennas, the time will scale with factor $\lceil \frac{A}{24} \rceil$ in the Small design and $\lceil \frac{A}{16} \rceil$ in the Large one. Note, however, that T_{ant} is different for the two cases, as discussed in the next section.

VI. RESULTS

Although the code that we developed is portable, we performed our experiments on a specific Xilinx FPGA target, the Virtex UltraScale+ used in the Amazon EC2 F1 instance (vu9p-flgb2104-2-i). This FPGA consists of 3 Super Logic Regions (SLRs) positioned at the left, middle, and right side of FPGA. It also contains 4 DDR4 memory interfaces with each interface accessing a 16 GiB memory. The middle SLR contains 2 memory interfaces while the left and right SLRs contain one interface each. Before reporting the performance results obtained on this FPGA, we briefly discuss the accuracy of the C++ code in comparison to the Acceleware code.

To perform the accuracy check, we simulated a grid of $50 \times 50 \times 50$ main cells with a boundary region of 10 cells on each side. Therefore, in total, the simulation space has $70 \times 70 \times 70 = 343000$ cells. The total number of time steps was 1000. Fig. 8 reports the magnitude of the S_{21} scattering parameter related to the transmission between an antenna source and an observation point in the simulation space as a function of frequency. This is a typical information used in the DBIM-TwIST MI algorithm and is obtained from the FDTD forward solver. The curves show an almost perfect overlapping between what Acceleware and our synthesizable C++ code obtain, with a Mean Absolute Percentage Error (MAPE) of 0.01%.

For what concerns the execution time, this is the product of the overall execution latency, in clock cycles, and the clock period. The latency is minimized at a high level with a proper design space exploration, which we could perform thanks to the flexibility of HLS coding and the features of Vivado HLS (2019.1 version). During the HLS phase, we aimed to keep the clock frequency target high enough so that the resulting performance would be competitive with the GPU design, but not too high in order to avoid issues at the implementation stage, specially during the routing phase. Determining the proper clock target and the most appropriate strategies for the implementation required a few iterations between the high-level abstract design in Vivado HLS and the low-level physical design in Vivado.

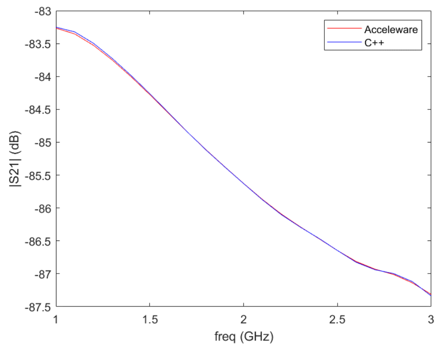


FIGURE 8. Accuracy comparison: Acceleware design versus our C++ code.

In the following we explain the impact of HLS-based optimization methods on the hardware performance. After that, we describe the design procedures and the results obtained first on a single FPGA and then on the multi-FPGA platform. We also report a comparison between the FPGA design and three GPU designs: the first one was developed in Matlab and tested on an NVIDIA Tesla K20c, the second one was developed using Acceleware and tested on a Tesla P40 GPU, and the third one is our design obtained from the same C++ code that runs on the FPGA and tested also on the Tesla K20c GPU. For the comparison we used the same simulation space of the accuracy check discussed above.

A. IMPACT OF HLS OPTIMIZATIONS ON PERFORMANCE

The HLS optimization techniques described in Sec. IV improve the performance of our FDTD hardware accelerator. It is important to note that different optimization strategies are applied step-by-step and the designer must have enough knowledge about the algorithm bottlenecks to find the best HLS optimizations. Each of these changes the performance and also the bottlenecks, hence the designer must monitor and analyze the performance change while exploring the accelerator design space using HLS. Here, the impact of each HLS optimization on the hardware performance is explored in more detail. Starting from the original code without any HLS directives, we add each optimization method incrementally and measure the performance of the FDTD Compute Unit (CU) in terms of latency and resource usage. The results in Fig. 9 show that each directive contributes to reducing the latency until the minimum latency of 8.6 s is obtained when all the directives are applied. (The figure is split in two histograms with different scales for better readability.) Fig. 10 shows the latency of each FDTD function and their relation with HLS directives. Each directive operates on one or multiple functions: *pipeline* and *inline* directives operate on all functions, storage for coefficients (*Coefs*) is applied to Update E and E-boundary functions, *Blocking* is applied to Update H and E, and loop *Merge* + storage for *Boundaries* is applied to H-boundary function.[2]

[2]It can be applied to E-boundaries as well, but it complicates routing in the implementation step in our target FPGA.

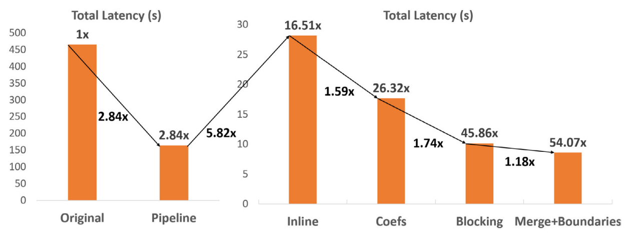


FIGURE 9. Impact of different HLS optimization methods on the total latency. (Numbers on top of the bars show the improvement compared to the original code, and numbers bellow the arrows show the improvement compared to the previous optimization method).

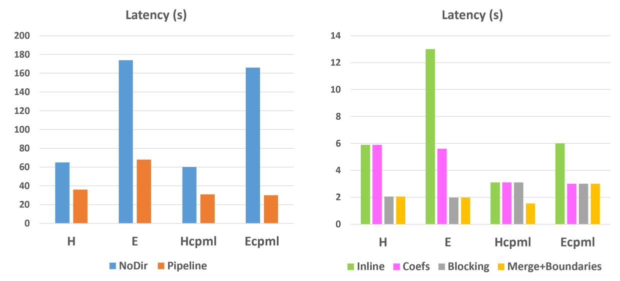


FIGURE 10. Impact of different HLS optimization methods on the latency of each FDTD function.

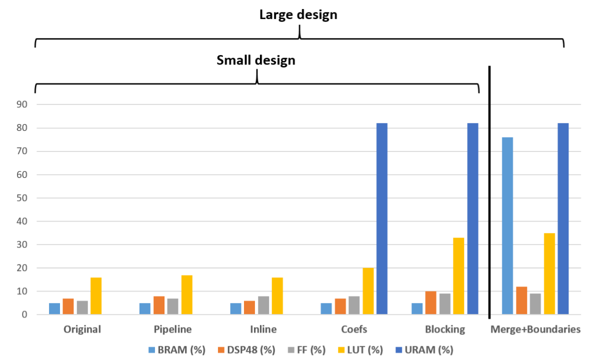


FIGURE 11. Impact of HLS optimization methods on resource usage per SLR.

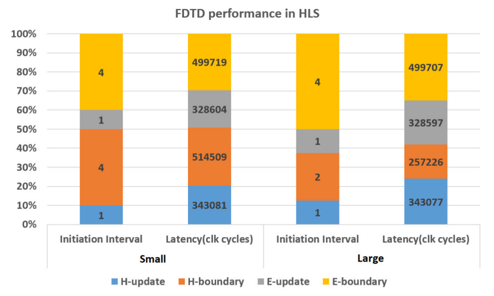


FIGURE 12. The performance of the main FDTD loops in Small and Large design in HLS.

The HLS estimation of the resource usage for the accelerator with respect to different HLS directives is shown in Fig. 11. Note that the Small design uses all the HLS directives except for loop merging and storage for the boundaries. The high URAM usage is due to the storage of constant coefficients (Sec. IV-D). The Blocking method (Sec. IV-B) increases the LUT usage and the last directive in the Large design (*Merge+Boundaries*) increases the BRAM usage. Although DSPs and FFs are not fully utilized, the advantage of using a large FPGA with high number of resources is the higher availability of URAMs. Compared to the design without URAMs (*Inline* directive in Fig. 11), the optimized Small design including URAMs (after *Blocking* method) obtains 2.8$\times$ improvement in the total latency (Fig. 9).

B. FDTD PERFORMANCE ON A SINGLE FPGA

To optimize the clock frequency, we used specific Vivado *strategies* for both logic synthesis and implementation. For synthesis we use the *Flow_PerfOptimized_High* strategy, which sets the tool options to maximize timing performance and gives less importance to minimizing resource usage (e.g., no resource sharing, FSM extraction forced to one-hot, no LUT combining, etc.). For the implementation of the Large design we used the *Performance_HighUtilSLRs* strategy, which aims to maximize the utilization of an SLR; for the Small one we used the *ExtraTiming_Opt* strategy, which gives priority chiefly to meeting timing constraints. For both designs, we obtain a maximum clock frequency of 167 MHz.

For the latency optimization, Fig. 12 shows II and latencies of the loops present in the various FDTD building blocks, for both the Small and Large designs. It is observed that for E Update and H Update we can achieve the minimum II of 1 thanks to the blocking method described in Sec. IV-B and the loop pipelining described in Sec. IV-D.

Optimizing the II of the boundary loops is a much more complicated task. First of all, many more variables need to be accessed from memory, as clear from the comparison between Algs. 3-4 and Alg. 2. Note that merging the loops on two boundary faces in the same direction, while beneficial for the sharing of some logic, does not reduce the number of accesses to variables defined over two physically distinct areas of the domain. Note also that further increasing the number of AXI ports is not possible because of the mentioned limitations of memory controllers and the complications arising from accessing multiple controllers in different SLRs. The only viable option is to store the E and H variables in local on-chip memories, while using four separate ports for Ψ_E and Ψ_H variables (the corresponding ports between E and H can be shared). Even with this solution, the best achievable II for the boundary loops with the selected number of AXI ports is 2, as determined by the simultaneous access from the same AXI port to variables defined in two distinct boundary faces.

Unfortunately, II=2 is not achievable for both E and H boundaries at the same time. In particular, sharing the same BRAMs for both boundary fields complicates routing, leading to timing failures in the implementation. An alternative is to use separate BRAMs, but the resource usage exceeds the available BRAMs in each SLR region (48% in total, i.e. about one and a half out of three SLRs available). As a result, one CU gets placed across two SLRs, and the routing phase in Vivado ends with a large negative slack because of the large routing delay of the many wires that cross the SLRs.

Nevertheless, by using the local memory either in E or H boundary loops, we respect the BRAM limits in one SLR without increasing the routing complexity. In the Large design, we apply it to H-boundary. This requires to have four separate ports for Ψ_{H*} (Din1-4 in Fig. 5 and Table 2 for the Large design). Therefore, as shown in Fig. 12, in the Large design the II for H-boundary is 2, while for E-boundary is 4. In the Small design, however, both E- and H- boundary loops obtain II=4 as the local memory is not used: eliminating ports Din2-4 already degrades II from 2 to 4, hence making local memory totally ineffective. Using less BRAMS, however, makes room for more CUs in a single FPGA, hence balancing the longer latency of each CU with higher parallelism.

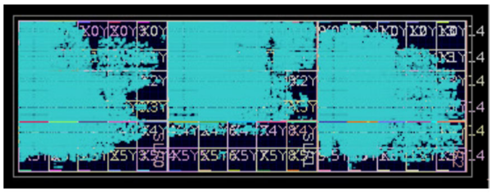
The clock frequency estimated by HLS for both design versions is 170 MHz, very close to the final 167-MHz frequency obtained after implementation. Critical paths are related to the data transfer from AXI ports to local memory (BRAM, LUT-based, or URAM). This is because memory blocks are scattered in the SLRs, thus causing significant routing delays.

The estimated execution time for the Small and Large designs is 10.14 s and 8.6 s, respectively, and is determined by the computation time and not by the DDR memory access. This is because the maximum bandwidth of 19.2 GB/s of each DDR4 memory bank in the AWS F1 instance is always greater than the peak required bandwidth given by simultaneous *writes* and *reads*, each of which consumes a bandwidth of 4B$\times$167 MHz = 0.667 GB/s. In the Small design, each CU uses one Memory Controller (MC) connected to a single memory bank for at most 15 reads and 4 writes (Alg. 4), thus the peak bandwidth is $(15+4)\times 0.667 = 12.7$ GB/s, which is less than 19.2 GB/s. In the Large design, each CU has 16 ports mapped to one MC (MC_1) and 2 ports to another MC (MC_2). With 16 reads and 4 writes in MC_1, and 2 reads and 1 write in MC_2, the peak bandwidth is $20 \times 0.667 = 13.36$ GB/s for MC_1 and $3 \times 0.667 = 2$ GB/s for MC_2, both less than 19.2 GB/s. When more CUs are mapped to one FPGA, it is not an issue either, as shown in the following.

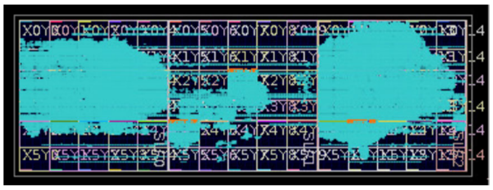
1) SINGLE FPGA SMALL DESIGN WITH 3 CUs

Since the Small design fits in one SLR and the FPGA consists of three SLRs, up to three CUs working in parallel on three antennas can be instantiated as shown in Fig. 13(a). Since each CU is connected to a separate MC and so to a separate memory bank, the DDR4 bandwidth limit is not exceeded.

Table 3 shows both the resource usage estimation obtained with Vivado HLS and the actual values after place-and-route in Vivado for the Small design. The HLS estimation includes only the FDTD block, while Vivado results include all the blocks in Fig. 4. The percentage for the HLS estimation in Table 3 refers to one SLR. For the Vivado results, it refers instead to the entire FPGA with three CUs, each using one single SLR. Table 3 shows a high resource utilization of URAMs. This is because we use them to store large constant arrays (a_{ei}, b_{ei}) in the update E function (see Sec. IV-C).



a) Small design (3 antennas)



b) Large design (2 antennas)

FIGURE 13. Device view in a) Small and b) Large design after place-and-route (it contains 3 SLRs in the left, middle and right side of the FPGA).

TABLE 3. Resource usage for the Small design: HLS estimation and Vivado implementation results.

		BRAM (%)	DSP48 (%)	FF (%)	LUT (%)	URAM (%)
HLS: 1 CU (One SLR)		4	11	10	35	82
Vivado: 3 CUs	FDTD	3.47	17.46	15.25	34.09	82
	Smart Connect	0	0	2.95	4.29	0
	DDR Ctrl	3.54	0.1	1.05	1.93	0
	Total	7	17.6	19.29	40	82

2) SINGLE FPGA LARGE DESIGN WITH 2 CUs

One CU in Large design consumes more BRAMs than the total number of BRAMs in one SLR. Therefore, a maximum of two CUs can be mapped onto one FPGA with three SLRs, as shown in Fig. 13(b). Two CUs share three MCs (MC_1, MC_2 and MC_3) while still not exceeding the DDR4 bandwidth. MC_1 and MC_2 are used for 2×16 ports and MC_3 is used for 2×2 ports. The peak bandwidth for both MC_1 and MC_2 is 13.36 GB/s (calculated as for the single CU). Regarding MC_3, the maximum number of reads and writes in 4 ports is 8 in the worst case leading to a peak bandwidth of $8\times 0.667 = 5.3$ GB/s, again less 19.2 GB/s.

The resource usage is shown in Table 4. The high BRAM usage is due to the local storage of the electric fields for the boundary region (*BF*, see Sec. IV-C). The percentage in the HLS estimation is for one SLR while the percentage in Vivado is for the entire FPGA. Note how two CUs use more than 2/3 of the total BRAM resources, hence making it impossible to map three CUs like in the Small design. For the blocking method described in Sec. IV-B we cannot map *Hram* in Alg. 2 to BRAMs, otherwise the design gets too congested (90% BRAM usage) and there is a significant penalty in clock frequency. For this reason we use LUTs as distributed RAM, using the LUT resource allocation pragma shown in Alg. 2.

As shown in Table 3 and Table 4, the DSP usage in both designs is low. This shows that our implementation is not compute-bound. The limiting factors in our design are related to the access to the internal on-chip memories and the number of AXI ports. We do not exceed the memory bandwidth by carefully selecting the number of AXI ports. Further increasing the number of ports (to the maximum supported ports that is $16 \times 4 = 64$), while still compatible with bandwidth, is not possible in practice due to the complexities in routing stage and accessing multiple SLRs. The application is memory-bound with respect to the access to on-chip memory. It is constrained by the internal resource limitations. To improve the performance, we need either more local storage, or a greater number of AXI ports. For the former we are limited by the internal resources, and for the latter we are restricted by the complexities arising from the routing stage.

TABLE 4. Resource usage for the Large design: HLS estimation and Vivado implementation results.

		BRAM (%)	DSP48 (%)	FF (%)	LUT (%)	URAM (%)
HLS: 1 CU (One SLR)		76	13	10	35	82
Vivado: 2 CUs	FDTD	73.7%	12.7	11.4	23.2	55
	Smart Connect	0	0	0.5	3.56	0
	DDR Ctrl	3.54	0.1	1.05	1.92	0
	Total	77.24	12.83	13.25	28.69	55

3) COMPARISON BETWEEN FPGA, GPU, AND CPU

Table 5 reports performance and power consumption for the FDTD accelerator on a CPU, three GPU designs (including the one derived from the code developed in this work), and FPGA. To allow a proper performance comparison between accelerators with different capacities in terms of parallel processed antennas, we chose the *time per antenna* as performance metric (total time divided by the number of parallel processed antennas). Since we could not measure the actual power consumed by the FPGA, the power consumption for FPGA reported in the third column of the table is the total consumed on-chip power obtained by the post-route report from the *Vivado Power Analysis* tool. This value can be therefore considered only a crude approximation of the actual power consumed by the device. The corresponding power consumption for CPU and GPU could not be measured either, so for a fair qualitative comparison, we compared the maximum Thermal Design Power (TDP) for our UltraScale+ FPGA with CPU and GPU designs in the second to last column of the table. In addition, we reported the maximum energy consumption in the last column (obtained from the TDP values and the execution time). It is observed that the maximum TDP for our FPGA target is between the CPU and GPU designs, but thanks to the lower execution time it would result in a more energy efficient implementation, should the actual power consumption scale more or less in the same way

TABLE 5. Performance comparison: CPU = Intel Xeon, GPU1 = Tesla K20C GPU2 = Tesla P40, GPU3 = GPU1 CUDA implementation, and FPGA (UltraScale+). TDP = Thermal Design Power, Energy = TDP×Time.

Hardware :release date	Max. ant.	Time per ant. (s)	On-chip Power (W)	Max. TDP (W)	Max. Energy (J)
CPU : 2017	1	24.97	-	105	2621
GPU1 (Matlab) : 2012	1	12.06	-	225	2713
GPU2 (Acceleware) : 2016	1	5.64	-	250	1410
GPU3 (this work, CUDA) : 2012	1	4.88	-	225	1098
UltraScale+ (This work, Small design) : 2016	3	3.38	16.24	128	432
UltraScale+ (This work, Large design) : 2016	2	4.3	14.5	128	550

in the three designs from TDP to actual power values. The results confirm that FPGAs can be more energy efficient than GPUs and CPUs, and show that FPGAs can be competitive with GPUs at scientific computation. Somewhat surprisingly, our GPU design (GPU3) is also 14% faster than the Acceleware GPU code. Compared to MATLAB implementation in GPU1, the CUDA implementation in this work (GPU3) is about 3 times faster. This is because CUDA is the native programming method for NVIDIA GPUs. As for the comparison between the two FPGA designs, the higher parallelism of the Small design results in a better performance, albeit at a higher power cost.

Table 6 shows a comparison between the performance of our proposed FPGA design for 3D FDTD with other FPGA implementations. It is observed that none of the previous works considered the impact of polarization currents which creates additional computations in each cell in the simulation space. In addition, the boundary conditions in our design is CPML in all directions that is not considered in works [26] and [19]. The performance can be measured in Mcells/s by dividing the total number of cells for all the time steps by the total processing time $((Max.Ant. \times T_{max} \times Total\_Cells)/Time(s))$. It can be converted to GFLOP/s by multiplying the performance in MCells/s to the number of operations in each cell. As shown in Table 6, although the performance in Mcells/s is not high in this work, the performance in GFLOP/s outperforms other implementations. This is because of the polarization currents in FDTD equations that create additional operations in each cell. Note that in [22], the peak performance was reported that is obtained without full CPMLs and is not a fair comparison with our work, so we computed the performance of [22] in MCells/s when there are CPML boundaries in all directions. In addition, our design can operate in higher clock frequency than previous methods.

TABLE 6. Performance comparison between our single Small FPGA design and other FPGA implementations.

	Work [26]	Work [19]	Work [22]	This work Small Design
Boundary Conditions	Dirichlet (zero)	CPML (limited)	CPML (Full)	CPML (Full)
Polarization	No	No	No	Yes
Language	MaxCompiler (HLS)	MaxCompiler (HLS)	Verilog	Vivado HLS (C++)
Mcells/s	325	100	336	101
GFLOP/s	**11.7**	**5.7**	**29.2**	**34.2**
Freq (MHz)	**100**	**100**	**100**	**167**

C. FDTD PERFORMANCE ON MULTIPLE FPGAs

Increasing the number of FPGAs can improve the FDTD performance. To assess such improvement we measured the execution time for a fixed number of antennas with different number of FPGAs. Fig. 14 shows the FDTD execution time for 8 and 24 antennas by varying the number of FPGAs from 1 to 10. It is observed that the reduction of the execution time depends on the number of antennas and FPGAs as well as the design version (Small or Large). Note from the curves in Fig. 14 that when the number of available FPGAs is sufficient to process all the antennas in parallel, increasing the number of FPGAs beyond that number does not further improve the performance. For example, in the Amazon EC2 F1 instance with 8 FPGAs, the Small and Large designs can process up to $A_{\max} = 3 \times 8 = 24$ and $A_{\max} = 2 \times 8 = 16$ antennas in parallel, respectively, in a fixed time equal to T_{ant} in Eqn. (6). Whenever the number of antennas to process exceeds $A_{\max}$, the time will increase according to Eqn. (6) with $F = 8$. Note how the best solution in Fig. 14, either Small or Large, depends on the number of antennas and the number of FPGAs, due to the interplay between the different number of CUs per FPGA and the different value of T_{ant}.

While the computation time remains constant for up to $A_{\max}$ antennas, the time required to transfer all the coefficients from the host to the DDR memories via the PCIe bus grows proportionally to the number of FPGAs because of the inevitable data duplication [30]. (As the initial values of E and H fields are zero, there is no need to take them into consideration.) To account for this overhead, we analyzed the maximum data transfer time. Table 7 shows the type and size of the coefficients used in Alg. 1 that need to be transferred. In Table 7, N is the size of the 3D FDTD simulation space including the boundary regions (70 × 70 × 70 in our experiments), and n_b is the size of the boundary in each side (10 in our case). Each coefficient, except Q_p, s_p, c_P, is a 3-dimensional vector. For example, b_h represents coefficients (b_{hx}, b_{hy}, b_{hz}).

TABLE 7. Dimensions of FDTD coefficients.

Coefficient	type	size	Coefficient	type	size
$a_h, d_h, c_p, s_p,$	scalar	1	$\mathbf{b_h, a_e, b_e, d_e}$	vector	$3N$
$c_{h1,2}, c_{e1,e2}$	vector	n_b	$\mathbf{Q_P}$	vector	N

The FDTD coefficients with the largest size in Table 7 are highlighted in bold. These coefficients are 3-dimensional vectors, except for Q_p, which is a 1-dimensional vector. Other coefficients have small size and do not affect the transfer time significantly. Therefore, the data to be transferred via PCIe amounts to $(4 \times 3+1) \times N = 13 \times N$ floating-point constants. By considering the maximum PCIe data transfer rate in the AWS F1 instance (12 GB/s), we can obtain the maximum data transfer time for each antenna, which is $T_{PCIe} = 1.4$ ms. By comparing T_{PCIe} with the total FDTD execution time for each antenna (8.6 s in the best case), we can see the data transfer time is negligible.

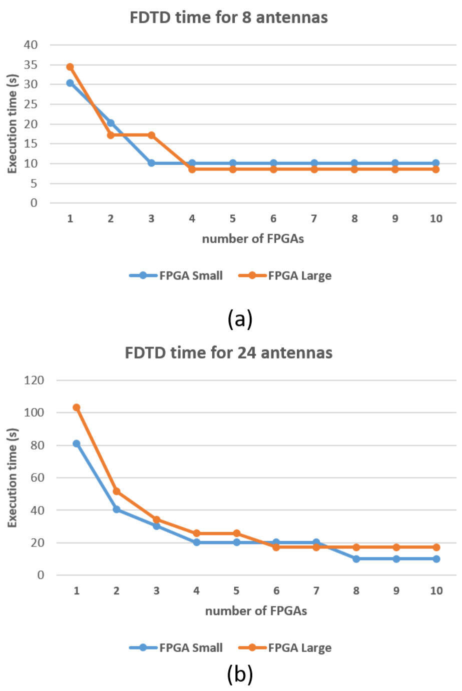


FIGURE 14. FDTD execution time for different number of FPGAs, (a) 8 antennas, (b) 24 antennas.

Fig. 15 shows the execution time for multiple antennas accelerated by 8 FPGAs and compares it with the best GPU results (Tesla k20c in this work, GPU3). With the exception of the cases in which one or two antennas are processed, the FPGA designs show always a higher performance. For many antennas to process, the multi-FPGA accelerator can significantly reduce the execution time. It should be noted that when the number of antennas is a multiple of 16 or 24, there is an increase in the execution time as predicted by Eqn. (6).

Finally, we report system-level performance results for the MI reconstruction algorithm shown in Fig. 1. With 24 antennas, on the Tesla P40 the execution time of the Acceleware FDTD code was 135.4 s per iteration, while the inversion part, executed by the host, is negligible (0.07 s). In our Small Multi-FPGA design, the FDTD takes instead 10.14 s, which corresponds to a speed-up of more than 13x. Compared to the GPU design in this work (GPU3) the speed-up is 11.5x.

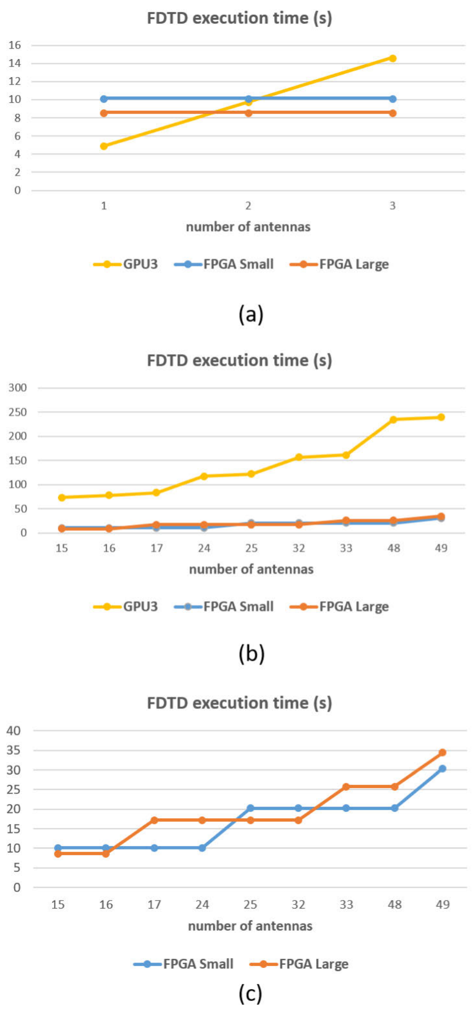


FIGURE 15. FDTD execution time for 8 FPGAs and different number of antennas, (a) from one antenna up to the maximum number in a single FPGA (3 for Small design), (b) Comparison of the single GPU in this work (GPU3, highly optimized for one antenna) and multi-FPGA design for multiple antennas, (c) more detailed view of multi-FPGA design results.

VII. DISCUSSION

In this section, we would like to summarize the main characteristics of our design of an FDTD accelerator, the main challenges faced during the design, and the solutions to these challenges by using HLS. First of all, it is important to note that although it is possible to take any synthesizable code written in C, C++, OpenCL, or systemC, and accelerate it in hardware by using HLS, the performance highly depends on the algorithm and it is not always possible to obtain the desired performance using HLS. It depends on the algorithm, degree of parallelization, hardware optimizations, target FPGA device, available resources, and achievable clock frequency.

Secondly, regarding the characteristics of our design, it should be noted that for our GPU design, all the resources are used to optimize the performance for one antenna. As the GPU resources are already fully used, there is no space left for further parallelization. On the contrary, in our FPGA design, we leveraged the multi-antenna parallelism to reduce the overall execution time. This is highly beneficial in Microwave Imaging systems due to the large number of required antennas. This parallelism on the number of antennas in addition to the number of cells in the 3D volume is one of the key characteristics of our FPGA design for FDTD algorithm. Other characteristics of our design are related to the design methodology that we adopted for the acceleration using HLS. Due to the iterative nature of FDTD algorithm, it is possible to parallelize the computations in the volume cells by ‘‘unrolling’’ and ‘‘pipelining’’ the loops. Merging the loops, local storage of boundaries and constant coefficients, and using a blocking strategy to reduce the memory access time are among other characteristics that could optimize the hardware acceleration.

Thirdly, we describe the challenges of the design and their solutions in this work. One of the main challenges in this application (3D FDTD) is the access to a high volume of data from external memory. This high memory bandwidth requirement becomes more challenging when we use the more advanced FDTD computations, related to the polarization currents (which has not been considered in previous works). To overcome this issue in this work, we used HLS capabilities to:

- efficiently process the extra computations on polarization currents by merging JP currents loops in Update E and E-boundary. Table 5 shows the higher GFLOPs for this work that is related to these extra computations.
- define high number of AXI ports to meet bandwidth requirement (Fig. 5, 15 or 18 ports)
- create a spatial blocking approach to reduce the data transfer time between external and local memories.

Another challenge is related to the computations in the boundary regions in 3D volume. Due to the access pattern in these regions, the parallelization of boundary computations is more challenging. By using HLS, we could:

- Merge the parallel loops in the boundary regions for parallel computations
- use local memories for boundary regions whenever possible to store the required data

There are some other challenges related to the hardware ‘‘implementation’’ stage when using more storage resources and higher number of ports. These configurations must be carefully selected in order to avoid routing failures in the implementation step. We proposed two hardware architectures (Large and Small) with different hardware configurations for a more flexible design, both of which are implementable in FPGA. The combination of these strategies with the usage of other HLS-based optimizations described in the paper makes it possible for our FPGA design to have a comparable performance to the GPU design and other HLS-based approaches.

Finally, there were several other HLS works tackling these problems for FDTD (and Stencils) acceleration in FPGA. Most of them focused on solving the issue of memory access time by presenting different blocking methods to process blocks of data from the 3D volume in multiple iterations. These methods include the combination of "spatial" and "temporal" blocking. Specifically, the OpenCL-based designs proposed in [21], [23], and [24] are among the successful works in the HLS domain for these blocking strategies. The problem of complex boundary conditions is still an open issue in hardware accelerators designed by HLS and the related works simplified the boundary conditions in favor of more parallelization. Using these methods in previous works could reduce the total processing time. However, ignoring the impact of polarization currents makes them ineffective in medical Microwave Imaging applications. The complex data dependencies created by the polarization currents call for a more straightforward approach to tackle the issues of memory transfer time, boundary conditions, and polarization currents at the same time, that is what we focused on in this work.

VIII. CONCLUSION

In this paper, we proposed a multi-FPGA hardware accelerator for 3D FDTD to be used in MI for medical applications. It is designed entirely in HLS making it possible to use several hardware optimization methods to obtain the best performance. The distinctive features of FDTD in this work are the modeling of polarization currents in dispersive materials, and the use of CPML boundary conditions in all directions. The combination of these features add extra complexity to the hardware design, but the HLS optimizations, including loop merge, blocking, pipelining and local memory storage, results in an efficient accelerator that is comparable with GPU or CPU-based design. Compared to the best GPU design of the same FDTD algorithm, a single FPGA can achieve 1.44x lower execution time per antenna. Our FPGA design is more energy efficient than CPU or GPU-based designs, and the maximum power for the FPGA design is still lower than the GPUs. In addition, the multi-FPGA design outperforms the other accelerators by processing multiple antennas in parallel. For a typical number of 24 antennas, 11.5x reduction of execution time can be achieved compared to the best GPU design.

REFERENCES

[1] D. O'Loughlin, M. O'Halloran, B. M. Moloney, M. Glavin, E. Jones, and M. A. Elahi, "Microwave breast imaging: Clinical advances and remaining challenges," *IEEE Trans. Biomed. Eng.*, vol. 65, no. 11, pp. 2580–2590, Nov. 2018.

[2] R. C. Conceição, J. J. Mohr, and M. O'Halloran, "Microwave tomograpghy," in *An Introduction to Microwave Imaging for Breast Cancer Detection*. Cham, Switzerland: Springer, 2016.

[3] M. R. Casu, M. Vacca, J. A. Tobon, A. Pulimeno, I. Sarwar, R. Solimene, and F. Vipiana, "A COTS-based microwave imaging system for breast-cancer detection," *IEEE Trans. Biomed. Circuits Syst.*, vol. 11, no. 4, pp. 804–814, Aug. 2017.

[4] J. A. T. Vasquez, R. Scapaticci, G. Turvani, G. Bellizzi, D. O. Rodriguez-Duarte, N. Joachimowicz, B. Duchêne, E. Tedeschi, M. R. Casu, L. Crocco, and F. Vipiana, "A prototype microwave system for 3D brain stroke imaging," *Sensors*, vol. 20, no. 9, p. 2607, May 2020.

[5] J. B. Schneider, *Understanding the Finite-Difference Time-Domain Method*. Pullman, WA, USA: Washington State Univ., 2017.

[6] Z. Miao and P. Kosmas, "Multiple-frequency DBIM-TwIST algorithm for microwave breast imaging," *IEEE Trans. Antennas Propag.*, vol. 65, no. 5, pp. 2507–2516, May 2017.

[7] A. Ltd. *FDTD Solvers*. Accessed: Feb. 13, 2021. [Online]. Available: https://www.acceleware.com/fdtd-solvers

[8] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1014–1029, May 2021.

[9] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? A study on the state of high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 5, pp. 898–911, May 2019.

[10] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2628–2639, Oct. 2020.

[11] Y.-K. Choi, Y. Chi, J. Wang, and J. Cong, "FLASH: Fast, parallel, and accurate simulator for HLS," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 4828–4841, Dec. 2020.

[12] L. Ferretti, J. Kwon, G. Ansaloni, G. D. Guglielmo, L. P. Carloni, and L. Pozzi, "Leveraging prior knowledge for effective design-space exploration in high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3736–3747, Nov. 2020.

[13] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Performance modeling and directives optimization for high-level synthesis on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 7, pp. 1428–1441, Jul. 2020.

[14] X. Wang, S. Liu, X. Li, and S. Zhong, "GPU-accelerated finite-difference time-domain method for dielectric media based on CUDA," *Int. J. RF Microw. Comput.-Aided Eng.*, vol. 26, no. 6, pp. 512–518, Aug. 2016.

[15] Z. Bo, X. Zheng-Hui, R. Wu, L. Wei-Ming, and S. Xin-Qing, "Accelerating FDTD algorithm using GPU computing," in *Proc. IEEE Int. Conf. Microw. Technol. Comput. Electromagn.*, May 2011, pp. 410–413.

[16] S. Liu, B. Zou, L. Zhang, and S. Ren, "Heterogeneous CPU+GPU-accelerated FDTD for scattering problems with dynamic load balancing," *IEEE Trans. Antennas Propag.*, vol. 68, no. 9, pp. 6734–6742, Sep. 2020.

[17] H. Zhang, Y. Lei, H. Ye, and Y. Gong, "Bistatic radar cross section prediction of 3-D target based on GPU-FDTD method," in *Proc. 12th Int. Symp. Antennas, Propag. EM Theory (ISAPE)*, Dec. 2018, pp. 1–4.

[18] T. Kenter, J. Forstner, and C. Plessl, "Flexible FPGA design for FDTD using OpenCL," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–7.

[19] H. Giefers, C. Plessl, and J. Förstner, "Accelerating finite difference time domain simulations with reconfigurable dataflow computers," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 5, pp. 65–70, Dec. 2013.

[20] Y. Takei, H. M. Waidyasooriya, M. Hariyama, and M. Kameyama, "FPGA-oriented design of an FDTD accelerator based on overlapped tiling," in *Proc. Int. Conf. Parallel Distrib. Process. Techn. Appl.*, 2015, pp. 72–77.

[21] H. M. Waidyasooriya, T. Endo, M. Hariyama, and Y. Ohtera, "OpenCL-based FPGA accelerator for 3D FDTD with periodic and absorbing boundary conditions," *Int. J. Reconfigurable Comput.*, vol. 2017, pp. 1–11, Aug. 2017.

[22] C. Kong and T. Su, "Parallel hardware architecture of the 3D FDTD algorithm with convolutional perfectly matched layer boundary condition," *Prog. Electromagn. Res. C*, vol. 105, pp. 161–174, 2020.

[23] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama, "OpenCL-based FPGA-platform for stencil computation and its optimization methodology," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 5, pp. 1390–1402, May 2017.

[24] H. R. Zohouri, A. Podobas, and S. Matsuoka, "Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 153–162.

[25] R. Soejima, K. Okina, K. Dohi, Y. Shibata, and K. Oguri, "A memory profiling framework for stencil computation on an FPGA accelerator with high level synthesis," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 4, pp. 69–74, Dec. 2014.

[26] K. Okina, R. Soejima, K. Fukumoto, Y. Shibata, and K. Oguri, "Power performance profiling of 3-D stencil computation on an FPGA accelerator for efficient pipeline optimization," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 4, pp. 9–14, Apr. 2016.

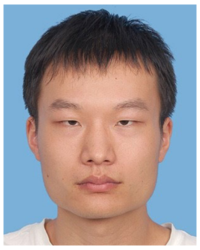
[27] S. Wang and Y. Liang, "A comprehensive framework for synthesizing stencil algorithms on FPGAs using OpenCL model," in *Proc. 54th Annu. Design Autom. Conf.*, Jun. 2017, pp. 1–6.

[28] K. Sano, Y. Hatsuda, and S. Yamamoto, "Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 695–705, Mar. 2014.

[29] K. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," *IEEE Trans. Antennas Propag.*, vol. AP-14, no. 3, pp. 302–307, May 1966.

[30] J. Shan, M. T. Lazarescu, J. Cortadella, L. Lavagno, and M. R. Casu, "CNN-on-AWS: Efficient allocation of multikernel applications on multi-FPGA platforms," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 2, pp. 301–314, Feb. 2021.

MOHAMMAD AMIR MANSOORI (Member, IEEE) received the B.S. and M.S. degrees in electronics engineering, in 2013 and 2016, respectively. He is currently pursuing the Ph.D. degree in electronics and telecommunications engineering with the Politecnico di Torino, Italy. His main research interests include FPGA-based embedded systems design, hardware/software co-design using high level synthesis, machine learning, and image processing. He is also a Marie-Curie Fellow and was a receiver of the MSCA-ITN Scholarship under the EMERALD Project.



PAN LU received the B.S. degree in computational mathematics from Sun Yat-sen University, China, in 2016, and the M.S. degree in advanced computing from King's College London, U.K., in 2017, where he is currently pursuing the Ph.D. degree in telecommunications. His research interests include signal processing, computational electromagnetics, and inverse scattering problems.

MARIO R. CASU (Senior Member, IEEE) received the Ph.D. degree in electronics and communications engineering from the Politecnico di Torino, Torino, Italy, in 2001. He is currently an Associate Professor with the Politecnico di Torino. His research interests include systems-on-chip with specialized accelerators, system-level design and design methodology for FPGAs and ASICs, and embedded machine learning. He is also interested in the design of circuits, systems, and platforms for industrial applications (biomedical, automotive, and food). His past work focused mostly on latency-insensitive design of systems-on-chip (SoC) and on networks-on-chip. He regularly serves in the technical program committee of international conferences, such as DAC, ICCAD, and DATE.